**实 验 报 告 （ 2 ）**

|  |  |
| --- | --- |
| **实验名称**：树的应用 | **实验地点**： 信息楼318 |
| **所使用的工具软件及环境：Win10, Java** | |
| **一、实验目的：**  1、掌握二叉树的结构特征，以及各种存储结构的特点及使用范围。  2、掌握用指针类型描述、访问和处理二叉树的运算。  3、掌握树的应用算法。 | |
| **二、实验内容描述：**（填写题目内容及输入输出要求）  （1）设计算法计算二叉树最大的宽度（二叉树的最大宽度是指二叉树所有层中结点个数的最大值）。  输入：创建的二叉树，先序排列，以#虚设结点。  输出：一个正整数N，代表着该二叉树的最大宽度。  （2）编写程序实现平衡二叉树AVL树。  输入：第一行为一个正整数N代表二叉树序列长度；第二行为N个数字，中间用空格分开，代表输入的N个二叉树顺序。  输出：生成的平衡二叉树序列，先序排列，以#表示空结点。  （3）已知一棵树的由根至叶子结点按层次输入的结点序列及每个结点的度  （每层中自左至右输入），试写出构造此树的孩子-兄弟链表的算法。  输入：一个正整数N结点数；然后输入N行，每行输入两个数字，中间用空格分开，代表节点及其对应的度。  输出：若该树有M个叶结点，则输出M行，每行是一条从根到叶子结点的路径，然后按照先序遍历的方式输出每一行。  第一题输入：ABC##DE#G##F##HI### 第一题输出：3  第二题输入：6 50 30 80 20 40 45 第二题输出：40 30 20 # # # 50 45 # # 80 # #  第三题输入：7 1 3 2 1 3 1 4 0 5 1 6 0 7 0 第三题输出： 1 2 5 7 1 2 3 6 1 2 3 4 | |
| **三、程序运行结果（说明设计思路，解释使用的数据结构，计算时间复杂度）**  **第1题**   1. 实验运行结果截图      1. 数据结构   定义二叉树结点的类：  public static class class TreeNode{  int value;  TreeNode right;  TreeNode left;  public TreeNode(int value){  this.value = value;  }  }  使用双端队列其中使用LinkedList作为对象，用于计算不为空的节点的个数。   1. 设计思路   自定义创建二叉树函数createTestTree和计算二叉树宽度函数widthOfBinaryTree。  ·先根据输入的先序遍历字符串创建二叉树，创建新的结点，并递归链接左右孩子结点，若遇到'#'，则向后遍历字符串；  public static TreeNode createTestTree(String s){ //根据输入的先序遍历字符串创建二叉树  TreeNode root = null;  if (s.charAt(i) != '#'){  //创建新的结点，并递归链接左右孩子结点  root = new TreeNode(s.charAt(i));  i++;  root.left = createTestTree(s);  root.right = createTestTree(s);  }else {  //若遇到'#'，则向后遍历字符串  i++;  }  return root;  }  ·再利用while循环从第一层开始把二叉树每一层不为空的节点放入队列，计算不为空的节点的个数。临时的节点来判断是否有左右子树，左右子树存在，则修改结点值入队，为下一层循环做准备；  public static int widthOfBinaryTree(TreeNode root) {  if (root == null) return 0;  Deque<TreeNode> queue = new LinkedList<>();  root.value = 0; //根节点存储的数据赋值为0，后面进行计算节点数  queue.offer(root);  int levelSize = 0;  while ( !queue.isEmpty()){  //从第一层开始把二叉树每一层不为空的节点放入队列  levelSize = queue.size();  ans = Math.max(ans, queue.getLast().value - queue.getFirst().value + 1);  //计算不为空的节点的个数  while ( levelSize-- != 0){  TreeNode node = queue.poll();//临时的节点来判断是否有左右子树  if (node.left != null){  //左右子树存在，则修改结点值入队，为下一层循环做准备  node.left.value = node.value\*2 + 1;  queue.offer(node.left);  }  if (node.right != null){  node.right.value = node.value\*2 + 2;  queue.offer(node.right);  }  }  }  return ans;  }  ·最后返回最大层数。   1. 时间复杂度   创建二叉树过程有一个while循环，时间复杂度为O(n)；计算二叉树宽度过程中有while循环，时间复杂度为O(n)，所以最终程序的时间复杂度为O(n)。  **第2题**   1. 实验运行结果截图      1. 数据结构   创建平衡树AVLTree和节点node的类，结构体如下：  class AVLTree {  private Node root;  //获取根节点、添加节点的方法、先序遍历…  }  class Node {  int value;  Node left;  Node right;  //返回子树的高度、返回当前节点的高度、左右旋转、添加节点的方法…   1. 设计思路   平衡树AVLTree和节点node里面设计add加入节点方法，加入节点后判断(左子树的高度-右子树的高度) 和1的关系，大于1，左子树左旋转，然后在根节点执行右旋转，反之右子树右旋转，根节点执行左旋转；  public void add(Node node){ //平衡树添加节点的方法  if(root == null){  root = node; //如果root为空直接让root指向node  } else {  root.add(node);  }  }  public void add(Node node){ //递归的形式添加节点，但需满足二叉排序树的要求  if(node == null){  return;  }  //判断传入的节点的值，和当前子树的根节点的值关系  if(node.value < this.value){  //如果当前节点，左子树为空null  if(this.left == null){  this.left = node;  } else {  //递归向左子树添加节点  this.left.add(node);  }  } else { //添加节点的值，大于等于当前节点值  //如果当前节点，右子树为空null  if(this.right == null){  this.right = node;  } else {  //递归向右子树添加节点  this.right.add(node);  }  }    //当添加完一个节点后，如果(右子树的高度-左子树的高度) > 1,左旋转  if(rightHeight() - leftHeight() > 1){  //如果当前节点的 右子树 的左子树高度 大于它的 右子树高度  if(right != null && right.leftHeight() > right.rightHeight()){  //当前节点的（右子树）进行右旋转  right.rightRotate();  }  leftRotate(); // 左旋转..  return; //必须要加  }    //当添加完一个节点后，如果：(左子树的高度-右子树的高度) > 1，右旋转  if(leftHeight() - rightHeight() > 1){  //如果当前节点的[左子树]的[[右子树]]高度大于它的[[左子树]]的高度  if(left != null && left.rightHeight() > left.leftHeight()){  left.leftRotate();//先对当前节点的（左子树）->（左旋转）  }  rightRotate(); //右旋转  }  }  ·其中节点高度为递归算法，左旋转函数把当前根节点拷贝作为左子树，拷贝的新节点指向根节点的左子树，将根节点右子树的左子树給拷贝节点，当前节点值改为右子树的，右子树上移作为根节点，把左子节点设置成新的节点，拷贝根节点，右旋转函数类似；  public int leftHeight(){//返回左子树的高度，返回右子树高度类似  if(left == null){  return 0;  }  return left.height();  }  public int height(){//返回当前节点的高度，以该节点为根节点的树的高度  return Math.max(  left == null ? 0 : left.height(),  right == null ? 0 : right.height()) + 1;  //左旋转的方法，右旋转类似  private void leftRotate(){  //把当前根节点拷贝作为左子树  Node newNode = new Node(value);  //拷贝的新节点指向根节点的左子树  newNode.left = left;  //将根节点右子树的左子树給拷贝节点  newNode.right = right.left;  //当前节点值改为右子树的  value = right.value;  //右子树上移作为根节点  right = right.right;  //把左子节点设置成新的节点，拷贝根节点  left = newNode;  }   1. 时间复杂度   创建平衡树的时间复杂度为O(lgn)；先序排列prefixOrder函数过程中，调用递归函数，时间复杂度为O(n)，所以最终程序的时间复杂度为O(lgn)。  **第3题**   1. 实验运行结果截图      1. 数据结构   创建树TreeNode类，结构体如下：  class TreeNode {  int value;  int nodenum;//所含子节点数  List<TreeNode> children;  public TreeNode(int value,int nodenum) {  this.value = value;  this.nodenum=nodenum;  children = new ArrayList<>();  }  }   1. 设计思路   由于输入是按顺序排列，所以依次放入数组arr，创建一棵树传入根节点，根据分支数依次加入节点，依次将每一层通过递归把每个节点的子树插入，利用cengshu记录当前层数，再进入下一层。  // 创建一棵树  int n=scanner.nextInt();  arr=new int[3\*n];  for (int j = 0; j <2\*n; j++) //把输入放进数组  arr[j]=scanner.nextInt();    while (arr[arrnum]!=0) {  bfs(nowNode,Cengshu);  Cengshu++;  }  ·其中insert函数传入参数树TreeNode和当前节点所含子节点数；bfs递归函数出口为当前层数为0，利用for循环把每个节点的子节点传入下一bfs进行递归；recurTree函数和findPath递归函数利用链表实现输出路径。  private static void bfs(TreeNode nowNode, int cengshu) {  if (cengshu==0) {  insert(nowNode, nowNode.nodenum);  return;  }  for (int i = 0; i <nowNode.nodenum; i++) {  bfs(nowNode.children.get(i), --cengshu);  cengshu++;  }  }  public static void recurTree(TreeNode root) { //recurTree函数，二维列表展示路径  List<List<Integer>> result =new ArrayList<>();  List<Integer> path = new ArrayList<>();  path.add(root.value);  findPath(result, root, path);  System.out.println(result);  }  //findPath递归函数，把每个节点的子节点加入到列表中，再进行下一节点的findpath递归  private static void findPath(List<List<Integer>> result, TreeNode node,List<Integer> path) {  if (node.children == null || node.children.size() <= 0){  result.add(path);  return;  }  for (int i = 0; i < node.children.size(); i++) {  TreeNode child =node.children.get(i);  List<Integer> cPath = new ArrayList<>();  cPath.addAll(path);  for (int j = 0; j < i; j++) {  TreeNode temchild =node.children.get(j);  cPath.add(temchild.value);  }  cPath.add(child.value);  findPath(result, child, cPath);  }  }   1. 时间复杂度   利用for循环完成输入，时间复杂度为O(n)；插入节点insert函数和bfs递归函数，findPath函数中有两个for循环嵌套，时间复杂度为O(n2)，所以最终程序的时间复杂度为O(n2)。 | |

2023年 5月 20日